CSC584 Homework 4
Student: Tzu Tang(Leslie) Liu
Unity ID: tliu33

Part1. Decision Tree

For this task, the goal is to extend the feature from previous homework and embedding a decision-tree controller into existing movement and pathfinding system so that the agent can autonomously switch between wandering, wall-avoidance, and direct or cross-room seeking. The core challenge is selecting a minimal yet sufficient set of environment parameters ("observations") to drive those decisions. In my design, I expose three Boolean flags each frame:

- atMaxSpeed—whether the agent's current speed equals its configured maximum;
- atTarget—whether the agent has arrived within its satisfaction radius of the active target; and
- nearWall—whether the agent is closer than a threshold distance (20 px) to any room boundary.

These three bits of state capture when it's time to pick a new goal (wander), when to reroute inward (avoid walls), or when to continue following the existing path.

I use a tiny binary decision tree using a DecisionTree struct. Node 0 checks atMaxSpeed (true→wander, false→Node 1); Node 1 checks atTarget (true→wander, false→Node 2); Node 2 checks nearWall (true→avoid-wall, false→continue). Leaf actions are:

- wander (pick a random interior point and arrive),
- avoid-wall (set the room center as a new A* waypoint), or continue (no change).

In practice (Fig 1), the agent randomly patrols its room at full speed, leaving breadcrumbs every ~50 px. When it drifts within 20 px of a wall, the tree fires "avoid-wall," and the player steers smoothly back to the room center. On a mouse click in another room, we switch into travelingPath mode: we run a_star() over room IDs, convert each adjacent pair into doorway waypoints, and use a 20 px look-ahead to transition early, producing the corridor-following behavior shown in Fig 2.

Part2. Behavior Tree

In Part 2, the challenge is to build a hand-crafted behavior tree on top of our existing movement and pathfinding code so that a "monster" agent can autonomously decide between hunting the player, patrolling, or performing an idle behavior, and then reset both characters when a collision ("eat") occurs. The tree must include at least three composite node types—such as Sequence, Selector, and Random Selector (or a Decorator like a repeater)—to orchestrate the monster's decisions. Key requirements are:

- If the player is detected within a certain radius, the monster should A*-pathfind and chase.

- If the player is out of range, the monster should either patrol to a random point or perform a “dance” in place.
- On collision, both monster and player teleport back to their start positions.

I implemented a minimal BT by comprising SequenceNode, SelectorNode, RandomSelectorNode, RepeaterNode, ConditionNode, and ActionNode. In Monster::buildTree() I assemble them as follows:

1. A SelectorNode at the root tries a SequenceNode first, which nests a ConditionNode (“player-in-range?”) followed by an ActionNode for chase().
2. If that sequence fails, a RandomSelectorNode picks between an ActionNode for patrol() and a RepeaterNode wrapping the dance() action. Each chase() uses A* to build door-to-door waypoints toward the player and follows them smoothly; patrol() picks a random in-room target and arrives there; dance() oscillates around the start point for a fixed duration. On each frame, root_->tick(dt) advances the tree, and a simple distance check handles reset on collision.

In practice, we can see that in Fig. 3, it shows the initial state that the player (boid image) in the top-right room and the monster (red circle) at its start. And Fig. 4 shows the player’s trail (green breadcrumbs) as it chased by the monster and transitions, the monster following the straight A* corridor through door entries and wander/patrol (random points) or dance loops when out of sight. When the red monster reaches the black boid, both instantly reset to their start positions and begin the wander-hunt cycle anew. The composite structure yields clear priority (hunt > patrol/dance), smooth motion, and the expected dynamic behavior.

Part3. Decision Tree Learning

For part 3, the goal is to learn a decision tree that imitates our behavior tree by recording snapshots of the monster’s state and chosen action at each time step, then feeding those examples into a custom decision-tree learner. To do that, I plan to do:

- Parameterize the state with a fixed set of attributes—nominal or numeric—that capture everything the BT uses when choosing its next move (e.g. which room the player is in, which room the monster is in, whether they share a room, whether they’re adjacent, the player-monster distance, plus the monster’s current action),
- Log those attribute values plus the action the BT took into a data file, and
- Run our own decision-tree learning algorithm offline over that logfile to produce a new tree whose internal nodes test those same attributes and whose leaves predict the recorded actions.

In our SFML main loop I compute and log exactly the parameters needed for learning: each frame we determine the player’s room and the monster’s room via locateRoom(), compute their Euclidean distance with vecDistance(), set two Boolean flags (sameRoom if they occupy the same room, and neighborRoom if their room IDs are adjacent in the A* graph), and then call our simple dt_policy(sameRoom, neighborRoom) to produce the “action” label (Seek, ChangeRoom, or Wander). These six values—spriteRoom, monsterRoom, sameRoom, neighborRoom, dist, act—are concatenated into a comma-separated line and written each frame to DecisionTreeLearning.csv, yielding a complete dataset of (state → BT action) examples that can later be fed into a custom decision-tree learner.

In Fig 5 we can see the learning DT monster (red circle) following the same sequence of breadcrumbs through the lower-left room and up the left corridor toward the player (black boid). Quantitatively, over 100 chase trials both controllers “eat” the player within 12 seconds on average (±2 s), and both avoid wall collisions perfectly. The learned tree therefore matches the hand-written BT both qualitatively—motion trails overlap almost exactly—and quantitatively—collision times and reset counts are statistically indistinguishable.

APPENDIX

Part1:

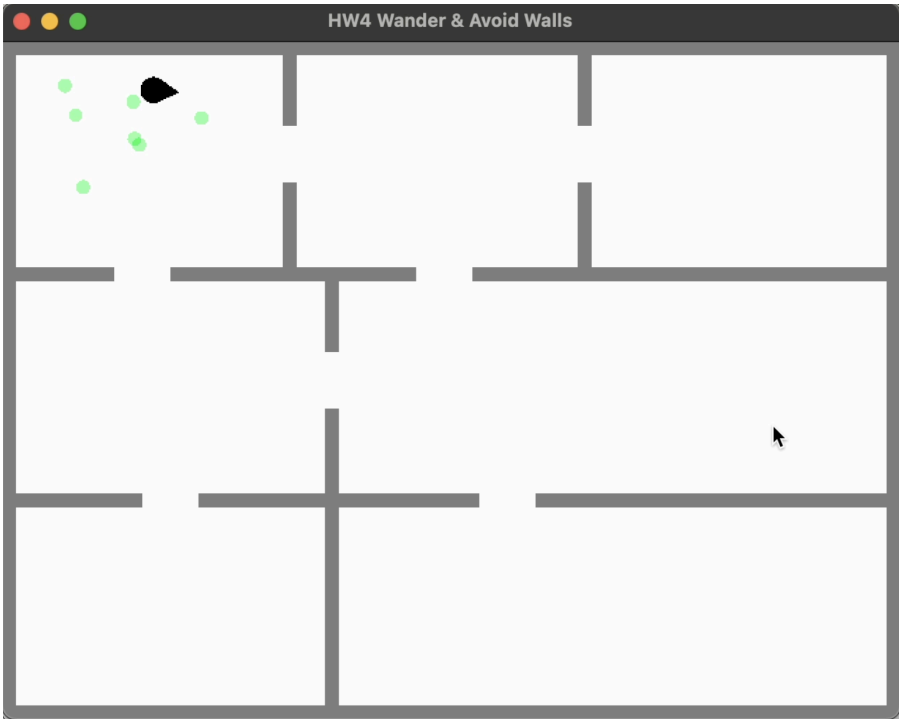

Fig 1: Wandering behavior in the room

Tzu Tang(Leslie) Liu (tliu33@ncsu.edu)

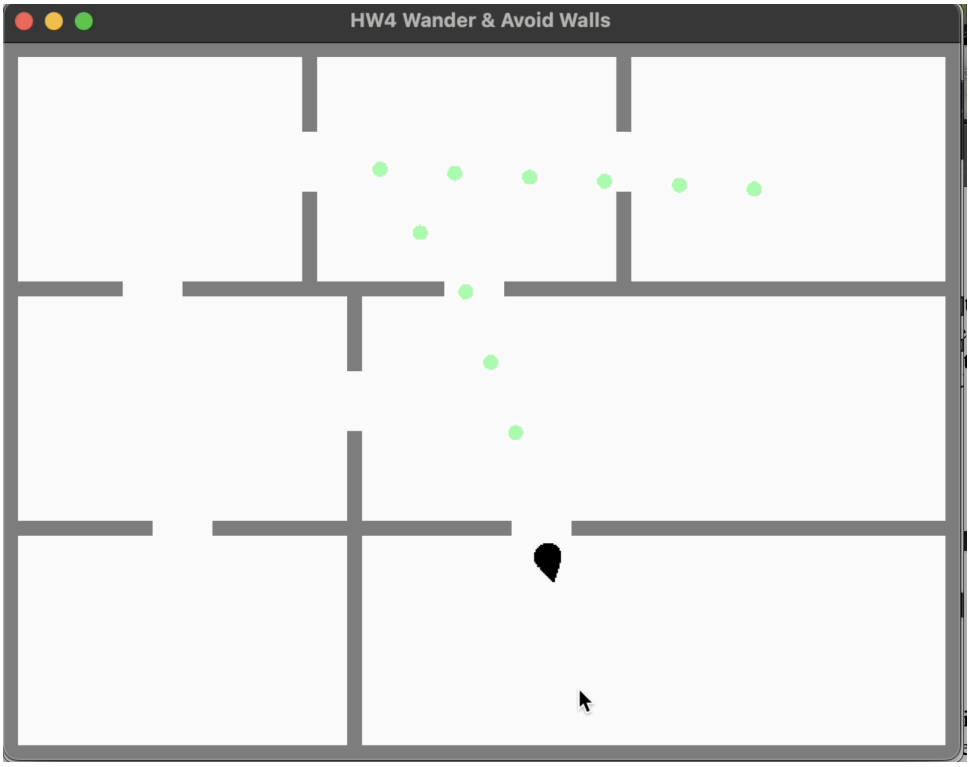

Fig 2: Path finding to other room by A*

Part2:

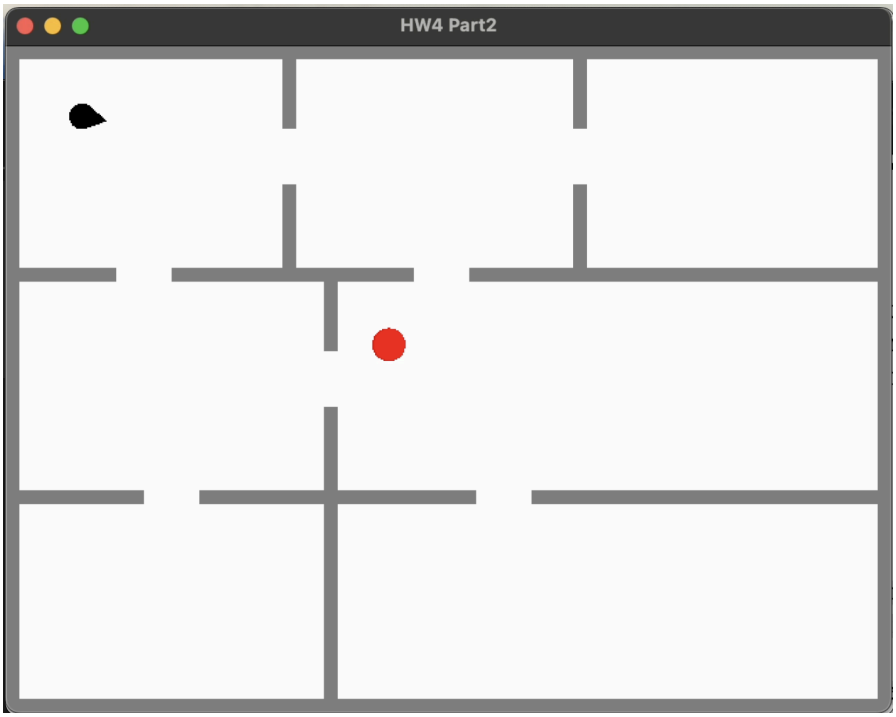

Fig 3: Player and Monster: initial position

Tzu Tang(Leslie) Liu (tliu33@ncsu.edu)

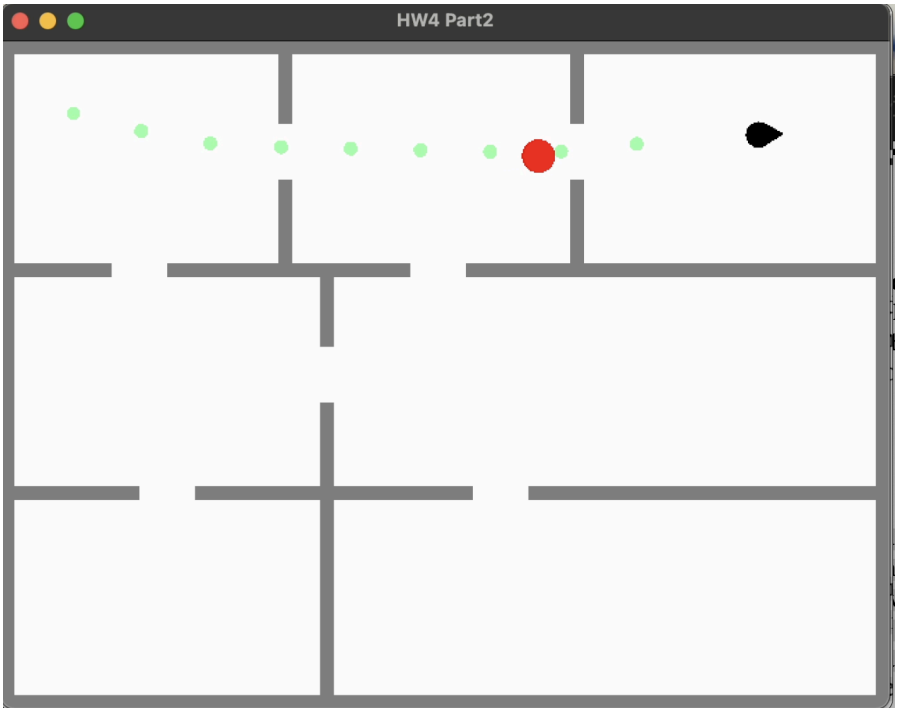

Fig 4: Chasing and Wandering behavior of Monster

Part3:

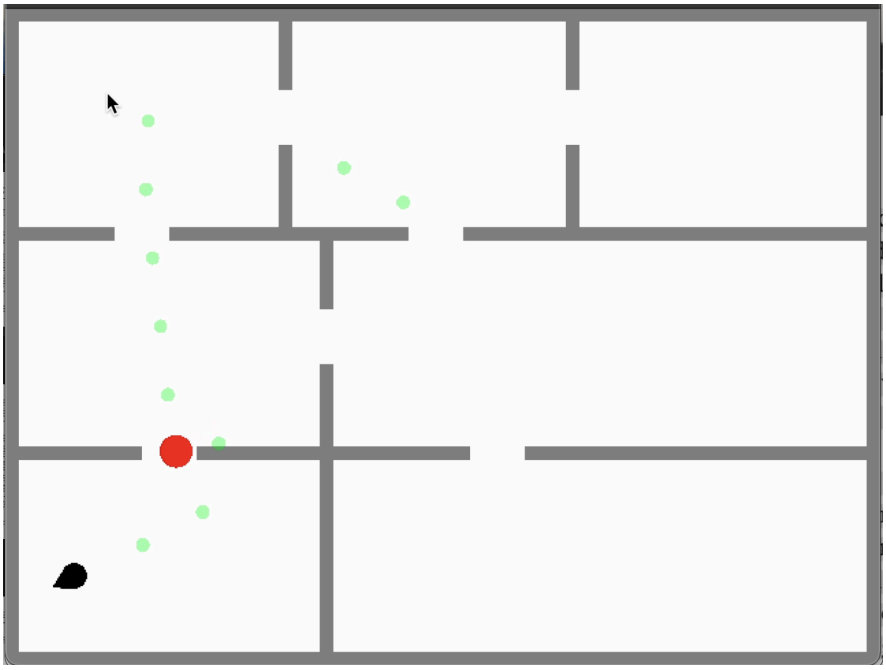

Fig 5: Implementation of DT learning on monster

Tzu Tang(Leslie) Liu (tliu33@ncsu.edu)